ATARI® PROGRAM EXCHANGE

John H. Palevich

DEEP BLUE SECRETS

Adapt the DEEP BLUE C COMPILER
to fit your own needs

Diskette: 48K (APX-20179)

User-Written Software for ATARI Home Computers

DEEP BLUE SECRETS

by

John Howard Palevich

Distributed By

The ATARI Program Exchange
P.O. Box 3705
Santa Clara, CA 95055

To request an APX Product Catalog, write to the address above, or call toll-free:

800/538-1862 (outside California)
800/672-1850 (within California)

Or call our Sales number, 408/727-5603

Trademarks of Atari

Printed in U.S.A.

Table of Contents

List of Figures

INTRODUCTION

OVERVIEW

DEEP BLUE C is an adaptation of Ron Cain's Small C-compiler for the ATARI Home Computer. DEEP BLUE C consists of three programs: the compiler, the linker, and the interpreter. The compiler is a modification of the original Small C compiler, published in "A Small C Compiler for the 8080's," Ron Cain, Dr. Dobb's Journal, #45 (May 1980), pp. 5-19. The linker and the interpreter are original works required to implement the C language on the 6502 microprocessor. With DEEP BLUE SECRETS, which is the source code for the DEEP BLUE C language and this manual, you can maintain, modify, and extend the language to fit your needs.

REQUIRED ACCESSORIES

48K RAM

ATARI 810 Disk Drive

DEEP BLUE C COMPILER (APX-20166)

ATARI Program-Text Editor(tm) and ATARI Macro Assembler (CX8121)

CONTACTING THE AUTHOR

Users wishing to contact the author about DEEP BLUE SECRETS may write to him at:

6200 Swords Way
Bethesda, MD 20817

THE DEEP BLUE C SYSTEM CODE DISKETTE

The Deep Blue C system consists of three programs:

1. A compiler that converts C source text (*.C) into compiled C code (*.CCC)

2. A linker that combines several compiled C code (*.CCC) files into a single executable object file (*.COM)

3. An interpreter that executes the object file (*.COM)

THE COMPILER FILES

CC*.C	The source code for the compiler
CC.LNK	The link file for the compiler

THE LINKER FILES

CLINK*.C	The source code for the linker
CLINKG.H	The global include file for the linker
CLINK.LNK	The link file for the linker

THE INTERPRETER FILES

THE INTERPRETER FILES

DBC*.MAC	ATARI Macro Assembler source text for the interpreter
MEDITMAC.ECF	PROGRAM.TEXT/EDITOR *.MAC customization file

THE COMPILER

Much of this chapter was taken from the public domain documentation file for Ron Cain's Small C compiler.

COMPILER SPECIFICATIONS

As of this writing, the compiler supports the following features:

Data type declarations can be:

*"char" (8 bits)

*"int" (16 bits)

by placing an "" before the variable name, a pointer can be formed to the respective type of data element

Arrays:

*single dimension (vector) arrays can be of type "char" or "int"

Expressions:

*unary operators:

+,-,*,&,++,--,!,$- (tilde)

binary operators:

+,-,*,/,%,!,↑,&,==,!=,<,<=,>,>=,<<,>>,op=,&&,||,?:,comma

primaries:

arrays[expression]

function(arg1,arg2,...,argn)

constant

* decimal number (69)
* octal number (0177)
* hexadecimal number (0xff)
* quoted string ("sample string")
* primed string ('a' or 'Z' or 'ab')
* local variable (or pointer)

* global (static) variable (or pointer)

Program control:

if,else,while,break,continue,return,for,do,switch,case,default
; (null statement)
$(statement; statement; ... statement;$)

Pointers:

local and static pointers can contain the address of "char" or "int" data elements

Compiler commands:

define name string (preprocessor will replace name by string throughout text)

include filename (allows program to include other files within this compilation)

Miscellaneous:

Expression evaluation maintains the same hierarchy as standard C.

Function calls are defined as any primary followed by an open parenthesis, so legal forms include:

```
*variable();

*array[expression]();

*constant();

*function()();
```

Pointer arithmetic takes into account the data type of the destination (e.g., pointer++ will increment by two if pointer was declared "int*pointer").

Pointer compares generated unsigned compares (since addresses are not signed numbers).

Generated code is "pure" (i.e., the code may be placed in Read Only Memory). Code, literals, and variables are kept in separate sections of memory.

The generated code is re-entrant. Every time a function is called, its local variables refer to a new stack frame. By way of example, the compiler uses recursive descent for most of its parsing, which relies heavily on re-entrant (recursive) functions.

LANGUAGE LIMITATIONS

Parts of the C language that are not supported are:

Structures

Multidimensional arrays

Floating point, long integer, or unsigned data types

Function calls returning anything but "int"

The unary "sizeof"

The binary type casting

The declaration specifiers "auto", "static", and "register"

The use of arguments within a "#define" command

COMPILER LIMITATIONS

Some limitations with the compiler are:

Since it is a single-pass compiler, undefined names are not detected and are assumed to be function names not yet defined. If this assumption is incorrect, the undefined reference will not appear until the compiled program is linked.

No optimizing is done. The code produced is sound and capable of re-entrancy, but no attempt is made to optimize either for code size or speed.

Constants are not evaluated by the compiler. That is, the line of code

```
X = 1+2
```

would generate code to add "1" and "2" at run time. The results are correct, but unnecessary code is the penalty.

STACK FRAME

Local variables and function arguments are kept on a 16-bit software stack that starts at the end of the global variable space and grows upwards. <u>This is the opposite of what the standard small-c stack does, and should be kept in mind when attempting to transfer compiler extensions from small-c to Deep Blue C.</u>

Function arguments are pushed onto the stack as they are encountered

between parentheses (note this is opposite that of standard C, which means routines expressly retrieving arguments from the stack rather than declaring them by name must beware). By the definition of the language, parameter passing is "call by value". Results are returned in the P register.

Local variables allocate as much stack space as is needed, and are then assigned the current value of the stack pointer (after the allocation) as their address.

It is worth pointing out local declarations allocate only as much stack space as is required, including an odd number of bytes, whereas function arguments always conist of two bytes apiece. In the event the argument was type "char" (8 bits), the most significant byte of the 2-byte value is 0.

The Deep Blue C stack discipline is fairly simple. Here is an example that shows almost every aspect of the stack:

Assume we are executing the following program:

```
main()$(
 char a;
 int b;
 char c[3];
 int d[2];
 f('A',0x1234);
$)

f(a,b)
 int a,b;
$(
 return;
$(
```

Here is the state of the abstract C machine stack just before the return statement in f() is executed. (By symmetry, it is also the state of the stack just after f() is called.)

Figure 3-1: Deep Blue C Stack Frame

```
      +-----------------------------+
$ffff |                             |
      +-----------------------------+
$fffe |         .    .    .         |
      |                             |
      +-----------------------------+
      |     high return address     |
      +-----------------------------+
      |      low return address     | <-- Stack Pointer
      +-----------------------------+
      |             $12             |
      +-----------------------------+
      |             $34             |
      +-----------------------------+
      |             $00             |
      +-----------------------------+
      |             $65             |
      +-----------------------------+
      |        high byte d[1]       |
      +-----------------------------+
      |         low byte d[1]       |
      +-----------------------------+
      |        high byte d[0]       |
      +-----------------------------+
      |         low byte d[0]       |
      +-----------------------------+
      |            c[2]             |
      +-----------------------------+
      |            c[1]             |
      +-----------------------------+
      |            c[0]             |
      +-----------------------------+
      |       high byte of b        |
      +-----------------------------+
      |        low byte of b        |
      +-----------------------------+
      |              a              |
      +-----------------------------+
      | end of code & global vars.  |
      +-----------------------------+
```

COMPILED C CODE FORMAT

The original small-c compiler produced asembly language source code. This approach, while easy to implement and maintain, was discarded because of the limited storage capacity of the ATARI 810 Disk Drive. Large C programs, such as the compiler itself, would have produced intermediate text files far larger than could be stored on an ATARI 810 Disk Drive.

So, to decrease the intermediate file size, a compiled c code format was invented. This format is nothing more than a preprocessed assembly language for the abstract C machine. *.CCC files are organized as a series of records, most of which have assembly-language counterparts.

$00-$7f

C machine opcodes. Equivalent of "db $xx<return>."

$80

LUSE - use of label. Followed by a word containing the label number. If the label number is less than 10000, then the label is a code label; otherwise, it is a variable label.

$81

LDEF - definition of label. Followed by a word containing the label number. If the label number is less than 10000, then the label is a code label; otherwise, it is a variable label.

$82

BCON - byte constant. Followed by a byte. This is equivalent to a "db $xx", and is used to ensure that the byte constant is not interpreted as a multibyte pseudo-op.

$83

WCON - word constant. Followed by a word. This is equivalent to a "dw $xx", and is used to ensure that a word constant is not interpreted as a multibyte pseudo-op.

$84

RDAT - random data. Followed by a word specifying an additional number of bytes that are part of the instruction. BCON and WCON could have been simulated by RDAT $0001 and RDAT $0002, respectively.

$85

LADR - literal address. Followed by a word specifying an offset into the string literal table. (The string literal table base address is always code label 1.)

$86

DSPC - define space. Followed by a word specifying how many bytes to reserve.

$87

LEXT - label external. Followed by a word specifying the variable number of a C text string containing the variable name.

$88

LGLB - label global. Followed by a word specifying the variable number and a C text string containing the variable name.

THE ABSTRACT C MACHINE

The linker produces code for an imaginary microprocessor called the abstract C machine. This imaginary processor is simulated by the interpreter, which executes 6502 machine-language subroutines to implement the abstract operations.

The C machine is a hybrid register/stack machine. The machine has one register, called P, a program counter, called PC, and a stack pointer, called SP. The stack grows up from the end of the user's code, and tests are performed to ensure that it does not overwrite the ATARI screen RAM, which grows downward, depending on graphics mode.

The abstract C machine recognizes less than fifty instructions; most are either one or three bytes long. Here is a list, in numerical order, of the opcodes:

$00

Assembly Language Escape. Followed by an address of a 6502 assembly language routine to call.

$01

Load P with byte absolute. Followed by address of byte.

$02

Load P with word absolute. Followed by address of word.

$03

Load P with address of local variable. Followed by offset from current stack pointer.

$04

Store P into byte absolute. Followed by address of byte.

$05

Store P into word absolute. Followed by address of word.

$06

Store P into byte at address on top of stack. Pop stack.

$07

Store P into word at address on top of stack. Pop stack.

$08

Load P with byte at address in P.

$09

Load P with word at address in P.

$0a

Reserved.

$0b

Load P with constant. Following word is constant.

$0c

Push P onto stack.

$0d

Test and jump if not zero - load PC with following address if and only if P does not contain zero.

$0e

Swap P and top of stack.

$0f

Call immediate. Followed by address of function to call, then a byte containing 2 + the number of arguments * 2. This value is subtracted from the stack pointer to remove the arguments and the return address. The value can also be used to determine the number of arguments passed to a function. See the source for printf(), in PRINTF.C for an example.

$10

Return - returns from a function call, adjusting the stack pointer as needed.

$11

Call top of stack - pops the address off the top of the stack and calls it.

12

Jump - loads PC with the following address.

$13

Test and jump if zero. Loads PC with the following address if and

only if P is zero.

$14

Adjust stack pointer. Adds the following word to SP. Used to obtain local variable storage.

$15

Double P. P<<=2;

$16

Add P and top of stack. P=(*SP--)+P;

$17

Subtract P from top of stack. P=(*SP--)-P;

$18

Multiply P by top of stack. P=(*SP--)*P;

$19

Divide top of stack by P. P=(*SP--)/P;

$1a

Remainder top of stack by P. P=(*SP--)%P;

$1b

Or P and top of stack. P=(*SP--)|P;

$1c

X or P and top of stack. P=(*SP--)↑P;

$1d

And P and top of stack. P=(*SP--)&P;

$1e

Shift top of stack right P times. P=(*SP--)>>P;

$1f

Shift top of stack left P times. P=(*SP--)<<P;

$20

Two's complement P. P=-P;

$21

One's complement P. P=$-P;

$22

Increment P. P=P+1;

$23

Decrement P. P=P-1;

$24

Test if top of stack equals P. P=(*SP--)==P;

$25

Test if top of stack does not equal P. P=(*SP--)!=P;

$26

Test if top of stack is less than P. P=(*SP--)<P;

$27

Test if top of stack is less than or equal to P. P=(*S--)<=P;

$28

Test if top of stack is greater than P. P=(*SP--)>P;

$29

Test if top of stack is greater than or equal to P. P=(*SP--)>=P;

$2a

Unsigned test if top of stack is less than P. P=(*SP--)<P;

$2b

Unsigned test if top of stack is less than or equal to P. P=(*SP--)<=P;

$2c

Unsigned test if top of stack is greater than P. P=(*SP--)>P;

$2d

Unsigned test if top of stack is greater than or equal to P. P=(*SP--)>=P;

$2e

Compare P to constant and jump if equal. Followed by word containing constant, and address to jump to. This is a five-byte instruction used to speed up the switch statement.

THE LINKER

The linker asks you for the name of a link file (*.LNK). This link file contains the names of all the files that make up the program you want to link. There are two kinds of files names in the link file:

1. *.CCC - compiled C code files. These files contain preprocessed abstract C machine assembly language, which the linker must assemble into abstract C machine code. They also contain references to external symbols that have to be resolved during the link phase.

2. *.OBJ - 6502 machine code files. One of these files, DBC.OBJ, contains the code for the abstract C machine interpreter. The rest of the *.OBJ files, if any, contain code used to implement assembly language subroutines.

The linker creates an executable object file (*.COM) in two passes. During the first pass, all *.OBJ files are ignored, and the *.CCC files are read to determine the value of all internal labels. After the first pass is completed, the linker resolves all external references in a link phase. If there are no errors in either the first pass or the link phase, the linker begins the second pass.

During the second pass, all *.OBJ files are copied as is directly to the *.COM file. The DBC.OBJ file has a standard DOS-II run address, so it should be the last file named in the *.LNK file. Other *.OBJ files are welcome to use the DOS-II init address so long as control is eventually returned to DOS II to contnue the loading process.

During the second pass, all *.CCC files are reread. This time, all abstract C machine opcodes are passed through to the output file, and all label-use (and literal-label-use) pseudo-ops have the correct value filled in.

The various *.CCC files are processed almost independently of each other. This means that global and string literal space is allocated for each *.CCC file at the end of that file's code segment. For instance, if the link file read:

```
D:ALPHA.CCC
D:BETA.CCC
D:DBC.OBJ
```

then the memory map would look like this:

Figure 6-1: C program memory usage

```
                         +----------------------------+
top of ram               |       screen display       |
MEMTOP+1                 |                            |
                         +----------------------------+
MEMTOP                   |         free space         |
SP+2                     |                            |
                         +----------------------------+
SP+1                     | Abstract C Machine Stack   |
SPORG                    |                            |
                         +----------------------------+
                         |      Globals for beta      |
                         +----------------------------+
                         | String literals for beta   |
                         +----------------------------+
                         | C Machine code for beta    |
                         +----------------------------+
                         |      Globals for alpha     |
                         +----------------------------+
                         | String literals for alpha  |
                         +----------------------------+
                         | C Machine code for alpha   |
                         +----------------------------+
BEGTOK+8                 |     high byte of SPORG     |
BEGTOK+7                 |     low byte of SPORG      |
                         +----------------------------+
BEGTOK+6                 |     high byte of main      |
BEGTOK+5                 |     low byte of main       |
                         +----------------------------+
BEGTOK+4                 |             0              |
BEGTOK+3                 | C Machine code rev. # (1)  |
BEGTOK+2                 |            'c'             |
BEGTOK+1                 |            'b'             |
BEGTOK ($4000)           |            'd'             |
                         +----------------------------+
$3fff                    |   C Machine Interpreter    |
$3000                    |                            |
                         +----------------------------+
$2fff                    |     ATARI DOS, OS, etc.    |
$0000                    |                            |
                         +----------------------------+
```

THE INTERPRETER

The interpreter is a fairly small program, called DBC.OBJ, that is loaded with every C program. This interpreter contains the 6502 machine code needed to perform the functions specified for the abstract C machine opcodes. In addition, DBC.OBJ also contains 6502 machine code that implements the I/O functions and other basic functions defined in the standard I/O library AIO.C.

If you are making extensive modifications to Deep Blue C, you might find it helpful to change the equate in the file DBC.MAC from:

```
debug     = 0     ;nz if debugging
```

to

```
debug     = 1     ;nz if debugging
```

When debug is set nonzero, the resulting DBC.OBJ file will contain several useful debugging features. Aside from doing more checking, you will be asked, at run time, if you'd like to see a trace of the abstract C machine's execution. The format of this trace is:

```
#WWWW pXXXX sYYYY jZZZZ <op>=NN

WWWW  Value of the program counter
XXXX  Value of the P register
YYYY  Value of the top of stack
ZZZZ  Value of the stack pointer
<op>  Four-character mnemonic (from the opcode
      table in DBC.MAC)
NN    Value of the opcode byte
```

RECOMPILING THE SOURCE CODE

All three of these programs can be recompiled on 48K, using one ATARI diskette, but in the case of the compiler, you'll have to transfer the *.CCC files to another diskette before you have enough room to link them.

RECOMPILING THE COMPILER

Compile CC0.C to CC9.C and CCV.C. Link these modules together using the file CC.LNK.

RECOMPILING THE LINKER

Compile CLINK.C, CLINK2.C, and CLINKD.C. Link these modules together using the file CLINK.LNK.

REASSEMBLING THE INTERPRETER

Load the ATARI Macro Assembler and give the command line:

```
D:DBC.MAC
```

DEEP BLUE C REFERENCE MANUAL

This chapter should be read in conjunction with Appendix A of The C Programming Language, by Brian W. Kernighan and Dennis M. Ritchie, 1978, Bell Telephone Laboratories, Inc. (published by Prentice Hall, Inc.). It is an attempt to formally define the Deep Blue C Language as a subset of the C language defined in that appendix. Sections of Appendix A will be referred to in square brackets. The first section, the "Introduction", is [1], and the last, the "Preprocessor", is [18,5].

[1-2.1] No change

[2.2] Identifiers (Names) No special restriction on external identifiers. all names are 8 characters, 2 cases.

[2.3] Keywords: The following identifiers are reserved for use as keywords and may not be used otherwise: int, char, extern, return, break, continue, if, else, for, do, while, switch, case, default, asm.

The following identifiers are not currently implemented, but are used by other C compilers: float, double, struct, union, long, short, unsigned, auto, register, typedef, static, goto, sizeof, entry, fortran.

[2.4] No change

[2.4.1-2.4.2] Integer constants: No long integer constants.

[2.4.3] Character constants: 'xx' defines a 16 bit constant with the first character as the most signifigant byte. The backslash escapes have been modified as follows: \f -- clear screen, \g -- ring bell, \h -- backspace, \n -- newline, \r -- delete line, \t -- tab, \\ -- backslash, \' -- single quote, \" -- double quote.

[2.4.4] Floating constants: Not implemented

[2.5] Strings: You can't continue strings across newlines. Don't include heart (control-comma) in a string because that character is used as an end-of-string marker.

[2.6] Hardware characteristics: For the ATARI 400/800: ATASCII, char -- 8 bits (unsigned), int -- 16 bits, pointers -- 16 bits.

[3] Syntax notation: Where implemented, it's the same.

[4] What's in a name: same, where implemented.

[5-6] No change.

[6.1] Characters are unsigned.

[6.2-6.3] Floating: Not implemented.

[6.4]	Pointers and integers: no change.
[6.5].	Unsigned:not implemented.
[6.6]	Arithmetic conversions: char converted to int, no other types are implemented.
[7,7.1]	Primary expression: Same where implemented.
[7.2]	Unary operators: Implemented: *,&,-,!,$- (tilde),++,--. Not implemented: casts, sizeof.
[7.3-7.15]	Multiplicative, additive, shift, relational, equality, bitwise, logical, conditional, assignment, and comma operators: No change.
[8,8.1]	Storage class specifiers: only explicit sc-specifier is extern.
[8.2]	Type specifier: only char and int.
[8.3]	Declarators: Very few forms are allowed: char c, int i, char *c, int *i, char c[], int c[], char c[10], int c[10].
[8.4]	Meaning of declarators: All functions return integers, only one dimensional arrays.
[8.5-8.8]	Structures, unions, initialization, type names, and typedef: not implemented.
[9.1-9.6]	Expression, block, conditional, while, do, and for statements: no change.
[9.7]	Switch statement: must have default clause, must have a break statement at the end of the block statement.
[9.8-9.10]	Break, continue, and return statements: no change.
[9.11-9.12]	Goto and labeled statements: not implemented.
[9.13]	Null statement: No change.
[10-10.2]	External definitions: O.K., but functions can only be of type int. Pointers can be returned as integers -- this is tacky, but works.
[11-11.2]	Scope rules: As implemented -- no static keyword.
[12-12.1]	Compiler control lines, token replacement: only the simple form of the #define directive is supported. #undef is not supported.
[12.2]	File inclusion: file name is normalized with a ".H" extension. <filename> is identical to "filename". #includes may not be nested.
[12.3-12.4]	Conditional compilation and line control; not implemented.

[13-14.4] Implicit declarations to explicit pointer conversions: as implemented.

[15] Constant expressions: Not implemented.

[16] Portability considerations: chars are unsigned, 'ab' has 'a' as the high byte, 'b' as the low byte, and is stored in RAM as <low> <high>.

[17] Anachronisms: old forms are not supported.

[18-18.5] Syntax summary: as implemented.

Limited Warranty on Media and Hardware Accessories. We, Atari, Inc., guarantee to you, the original retail purchaser, that the medium on which the APX program is recorded and any hardware accessories sold by APX are free from defects for thirty days from the date of purchase. Any applicable implied warranties, including warranties of merchantability and fitness for a particular purpose, are also limited to thirty days from the date of purchase. Some states don't allow limitations on a warranty's period, so this limitation might not apply to you. If you discover such a defect within the thirty-day period, call APX for a Return Authorization Number, and then return the product along with proof of purchase date to APX. We will repair or replace the product at our option.

You void this warranty if the APX product: (1) has been misused or shows signs of excessive wear; (2) has been damaged by use with non-ATARI Home Computer products; or (3) has been serviced or modified by anyone other than an Authorized ATARI Computer Service Center. Incidental and consequential damages are not covered by this warranty or by any implied warranty. Some states don't allow exclusion of incidental or consequential damages, so this exclusion might not apply to you.

Disclaimer of Warranty and Liability on Computer Programs. Most APX programs have been written by people not employed by Atari, Inc. The programs we select for APX offer something of value that we want to make available to ATARI Home Computer owners. To offer these programs to the widest number of people economically, we don't put APX products through rigorous testing. Therefore, APX products are sold "as is," and we do not guarantee them in any way. In particular, we make no warranty, express or implied, including warranties of merchantability and fitness for a particular purpose. We are not liable for any losses or damages of any kind that result from use of an APX product.

ATARI®
PROGRAM
EXCHANGE

P.O. Box 3705
Santa Clara, CA 95055

Review Form

We're interested in your experiences with APX programs and documentation, both favorable and unfavorable. Many of our authors are eager to improve their programs if they know what you want. And, of course, we want to know about any bugs that slipped by us, so that the author can fix them. We also want to know whether our instructions are meeting your needs. You are our best source for suggesting improvements! Please help us by taking a moment to fill in this review sheet. Fold the sheet in thirds and seal it so that the address on the bottom of the back becomes the envelope front. Thank you for helping us!

1. Name and APX number of program.

____

____

2. If you have problems using the program, please describe them here.

____

____

____

3. What do you especially like about this program?

____

____

____

4. What do you think the program's weaknesses are?

____

____

____

5. How can the catalog description be more accurate or comprehensive?

____

____

6. On a scale of 1 to 10, 1 being "poor" and 10 being "excellent", please rate the following aspects of this program:

_______ Easy to use
_______ User-oriented (e.g., menus, prompts, clear language)
_______ Enjoyable
_______ Self-instructive
_______ Useful (non-game programs)
_______ Imaginative graphics and sound

7. Describe any technical errors you found in the user instructions (please give page numbers).

8. What did you especially like about the user instructions?

9. What revisions or additions would improve these instructions?

10. On a scale of 1 to 10, 1 representing "poor" and 10 representing "excellent", how would you rate the user instructions and why?

11. Other comments about the program or user instructions:

From

STAMP

ATARI Program Exchange
P.O. Box 3705
Santa Clara, CA 95055

[seal here]